FICHA 10
ÁRVORES BINÁRIAS DE PESQUISA

10.1. Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de árvore binária de pesquisa;
- Saber manipular as árvores, particularmente no que diz respeito à implementação de métodos de inserção/remoção de elementos e travessia das mesmas.

10.2. Introdução às árvores binárias de pesquisa

As árvores binárias são estruturas adequadas à representação de dados que obedeçam implicitamente a uma ordem hierárquica diádica, por exemplo a relação familiar entre um alguém e os seus ascendentes ou descendentes. Há ainda situações em que apesar de os dados não exibirem uma tal estruturação, determinadas operações que se pretendem efetuar sobre os mesmos podem ser muito facilitadas se estes forem armazenados numa árvore binária, por exemplo a pesquisa de conjuntos ordenados.

Na Figura 1, ilustra-se uma árvore binária destinada ao armazenamento de inteiros.

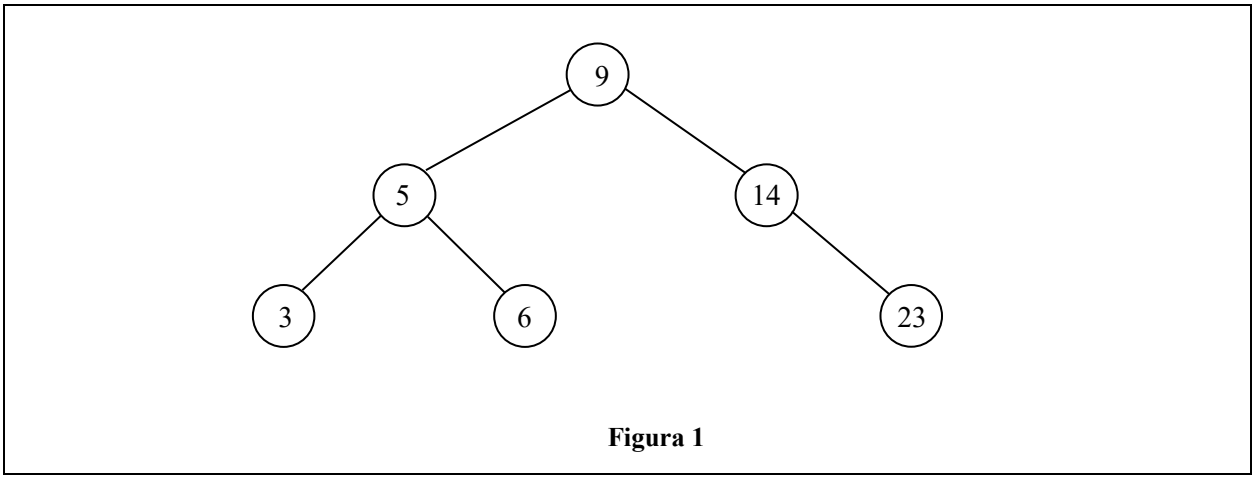


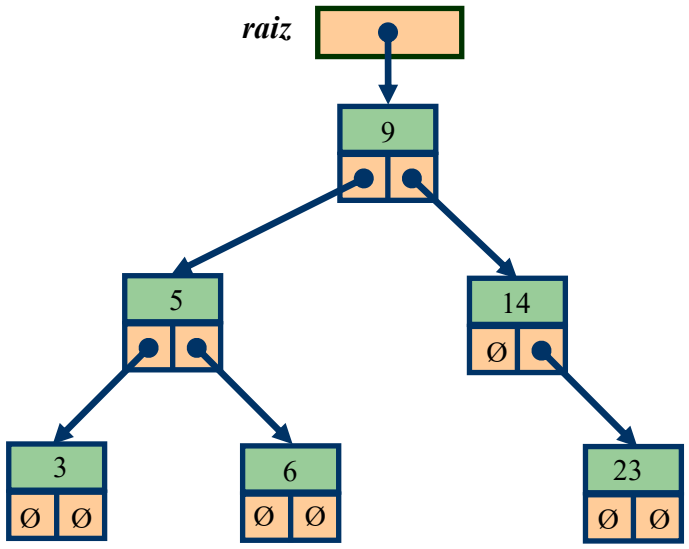
Figura 1

Repare na magnitude relativa dos elementos e correspondente posição na árvore. Verificará certamente que todos os elementos que constituem a subárvore esquerda de um determinado nó, caso ela exista, têm valor inferior ao valor desse nó. Analogamente, todos os elementos que constituem a subárvore direita de um determinado nó, caso ela exista, são maiores que o valor desse nó. Trata-se de uma **árvore binária de pesquisa.**

Há várias formas de implementar árvores binárias de pesquisa (e árvores em geral). Pode-se por exemplo optar por uma estrutura baseada numa tabela de elementos em que cada elemento representa um nó da árvore. Não obstante este tipo de representação ter algumas vantagens, neste documento trataremos apenas de uma forma alternativa baseada em ponteiros e variáveis dinâmicas, portanto uma estrutura de dados ligada, que passamos a descrever.

Na representação baseada em ponteiros e variáveis dinâmicas, cada nó é representado por uma *classe de objetos* que, para além de ter espaço reservado à informação, terá também um ponteiro para a subárvore esquerda e um ponteiro para a subárvore direita. Em certas situações, é vantajoso que esta estrutura de dados inclua também um ponteiro para o nó ascendente (nó pai). Por uma questão de simplicidade, não contemplaremos aqui essa possibilidade.

Com uma representação deste tipo, a árvore da Figura 1 teria o aspeto que se ilustra na Figura 2.



Legenda : Ø – Ponteiro Nulo

Figura 2

Para representar os nós de uma árvore com estas caraterísticas, podemos definir a classe `CNoArvore`:

```
class CArvoreBinaria; // "declarada" aqui para poder ser designada como amiga
                      //     antes de ser definitivamente declarada mais à frente

class CNoArvore {
  int dados;
  CNoArvore *esq; // ponteiro para subárvore esquerda
  CNoArvore *dir; // ponteiro para subárvore direita
  friend class CArvoreBinaria; // esta classe pode aceder a atributos private
};
```

Esta classe possui um atributo inteiro, `dados`, que representa a informação a armazenar (neste caso um inteiro) e dois atributos do tipo ponteiro, `esq` e `dir`, que permitem aceder às subárvores esquerda e direita, respetivamente.

A árvore propriamente dita, pode ser definida por uma classe semelhante à seguinte:

```
class CArvoreBinaria {
  CNoArvore *raiz;
  ...
  public:
  CArvoreBinaria();
  ~CArvoreBinaria();
  ...
};
```

Neste caso, o construtor tem como finalidade guardar no ponteiro `raiz` um valor (um endereço) consistente com o facto de inicialmente a árvore se encontrar vazia, ou seja, atribui a `raiz` o valor `nullptr`.

```
// Construtor
CArvoreBinaria::CArvoreBinaria() {
  raiz = nullptr;
}
```

O destrutor tem por fim "apagar" completamente a árvore e terá de efetuar a seguinte sequência de operações:

1. Se `raiz==nullptr` não há nada a fazer.
2. Caso contrário:
    a. Apaga subárvore esquerda,
    b. Apaga subárvore direita,
    c. Apaga nó atual.

Uma implementação possível é a que se apresenta a seguir, que foi construída com base num método que pode ser utilizado explicitamente para destruir toda a informação guardada na árvore, i.e. todos os nós:

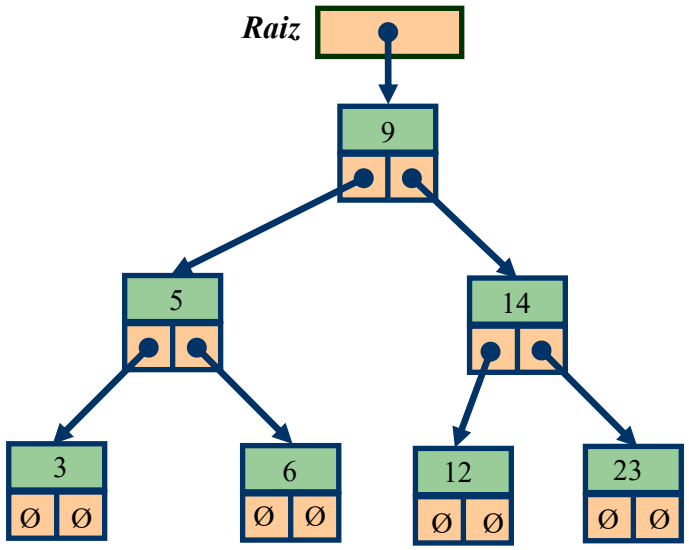
```
// Destrutor
CArvoreBinaria::~CArvoreBinaria() {
  destroi(); // método que elimina todos os nós
}
//-----------------------------------------------------------------------------
// Método public para destruir a árvore completa
void CArvoreBinaria::destroi() {
  destroi(raiz);
  raiz = nullptr;
}
//-----------------------------------------------------------------------------
// Método private e recursivo (apaga árvore "apontada" por 'pArvore')
```

```
void CArvoreBinaria::destroi(CNoArvore *pArvore) {
  if(pArvore != nullptr) {
    destroi(pArvore->esq);
    destroi(pArvore->dir);
    delete pArvore;
  }
}
```

Acrescentando estes métodos, a declaração da classe `CArvoreBinaria` passa a ser:

```
class CArvoreBinaria {
  // Dados e métodos privados
  CNoArvore *raiz;
  void destroi(CNoArvore *);
public:
  // Dados e métodos públicos
  CArvoreBinaria();
  ~CArvoreBinaria();
  void destroi();
};
```

Tente perceber as razões da necessidade de existência destes vários métodos e seus tipos de visibilidade (`public` ou `private`).



Legenda : Ø – Ponteiro Nulo

Figura 3

Uma estrutura de dados só tem alguma utilidade se for possível inserir nela elementos (dados). Temos assim que criar métodos para inserir novos dados, i.e. novos nós, na árvore binária. Vamos aqui tratar

apenas da inserção em árvores binárias ordenadas. Nessa situação, para inserir um elemento na árvore, é necessário um método que atravesse a árvore até atingir o "local" correto onde o novo elemento deve ser inserido.

Tomando como exemplo a árvore da Figura 2, se pretendermos inserir o elemento **12**, como este é maior que **9**, terá que fazer parte da subárvore direita. Sendo **12** menor que **14**, o novo elemento pertencerá à subárvore esquerda de **14**. Como, porém, esta subárvore está vazia, o novo elemento passará a ser a subárvore esquerda de **14**, tal como se vê na Figura 3.

Há uma situação particular que merece alguma reflexão. Neste exemplo, o elemento a inserir não estava presente na árvore. O que fazer se o elemento que pretendemos inserir já existe na árvore? Há duas soluções; ou se aceita que possam existir elementos duplicados, ou aquando de uma tentativa de inserção de um elemento já existente não se efetua a inserção. A escolha entre estas duas alternativas depende da aplicação do programa. No caso presente, optou-se pela segunda, ou seja só se procede à inserção se o valor a inserir ainda não existir na árvore.

Um método adequado para inserir um elemento inteiro de forma ordenada na nossa árvore binária é apresentado na caixa sequente. Também aqui se optou por separar o método num "front-end" público mantendo a "maquinaria" na parte privada.

```
// Metodo public chamado para fazer a inserção
void CArvoreBinaria::insere(int item) {
    if (raiz != nullptr) insere(item, raiz); // chama o metodo private
    else {  // cria o primeiro no da árvore
         raiz = new CNoArvore;
         raiz->dados = item;
         raiz->esq = raiz->dir = nullptr;
     }
}
//---------------------------------------------------------------------
// Método private que "faz o trabalho"
void CArvoreBinaria::insere(int item, CNoArvore *raiz) {
     if (item < raiz->dados) {
         if (raiz->esq != nullptr) insere(item, raiz->esq);
         else { // novo nó é o primeiro nó da subárvore esquerda
             raiz->esq = new CNoArvore;
             raiz->esq->dados = item;
             raiz->esq->esq = raiz->esq->dir = nullptr;
         }
     }
     else if (item > raiz->dados) {
         if (raiz->dir != nullptr) insere(item, raiz->dir);
         else { // novo nó é o primeiro nó da subárvore direita
             raiz->dir = new CNoArvore;
             raiz->dir->dados = item;
             raiz->dir->esq = raiz->dir->dir = nullptr;
         }
     }
}
```

A declaração da classe `CArvoreBinaria` atualizada com as declarações destes dois métodos é:

```
class CArvoreBinaria {
  CNoArvore *raiz;
  void destroi(CNoArvore *);
  void insere(int, CNoArvore *);
public:
  CArvoreBinaria();
  ~CArvoreBinaria();
  void destroi();
  void insere(int);
};
```

Por vezes é necessário pesquisar um determinado elemento numa árvore binária. Se a intenção for apenas determinar se esse elemento faz ou não parte da árvore, um método adequado devolve como resultado apenas um valor booleano indicativo da presença ou ausência do elemento. Em algumas situações, é importante devolver um ponteiro para o nó que contém o valor procurado. Na caixa seguinte, apresentam-se dois métodos que permitem fazer uma pesquisa da árvore e devolver um booleano. Também aqui existem dois métodos, sendo um o "front-end" público e outro o método `private` responsável pela pesquisa da árvore. Pense como o poderia alterar facilmente o código para devolver um ponteiro para o nó que contém o valor procurado.

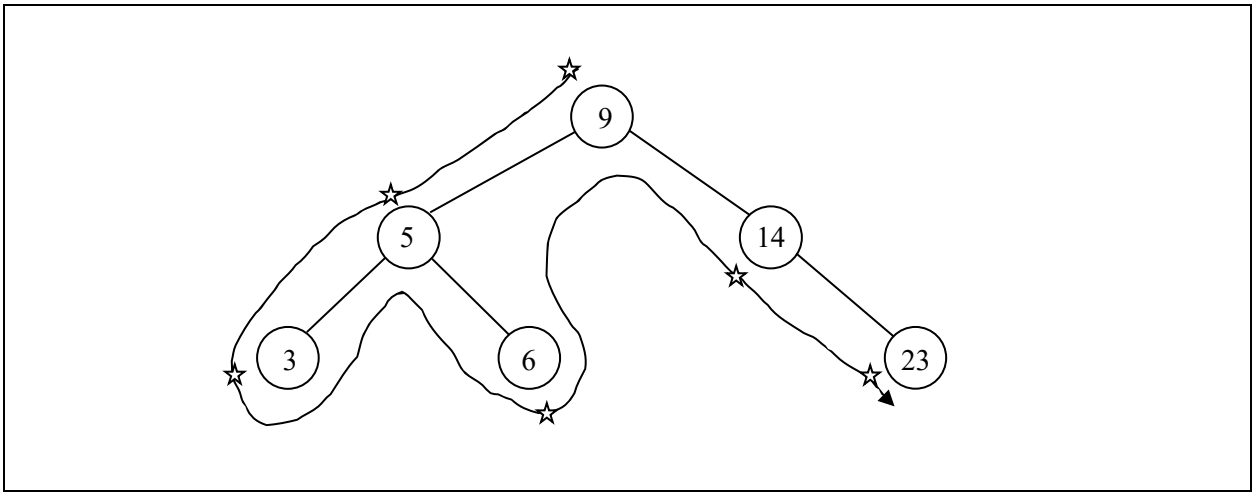
```
// Front-end public
bool CArvoreBinaria::procura(int item) const {
   return procuraRecursiva(raiz, item);
}
//-----------------------------------------------------------------------

// Método private
bool CArvoreBinaria::procuraRecursiva(CNoArvore *raiz, int item) const {
     // casos elementares
     if (raiz == nullptr) return false;
     if (raiz->dados == item) return true;

     // caso geral
    if (raiz->dados < item)
    return procuraRecursiva(raiz->dir, item);
    else return procuraRecursiva(raiz->esq, item);
}
```

Como sabe, listar todos os elementos de uma estrutura de dados linear, como as tabelas e listas, é uma tarefa trivial. O mesmo não se passa com as árvores pois, devido à sua estrutura, não possuem uma ordem natural. Com efeito, e tomando por exemplo a árvore binária da Figura 1, é possível mostrar o conteúdo da árvore segundo várias ordens.

Segundo a ordem **Pré-Ordem,**

Pré-Ordem
1. Lista elemento corrente, 2. Lista elementos da subárvore esquerda, 3. Lista elementos da subárvore direita.

como se ilustra a seguir,



onde a linha mostra o trajeto da travessia e as estrelas mostram em que ponto do trajeto o elemento adjacente é listado. Segundo esta ordem, os elementos seriam listados na sequência 9, 5, 3, 6, 14, 23. A listagem em pré-ordem é a adequada para guardar árvores binárias ordenadas em ficheiros, para se poder depois reproduzir a estrutura da árvore aquando da leitura desses ficheiros.

É possível definir outras duas ordens de travessia: a ordem **Por-Ordem** e a ordem **Pós-Ordem.**

Por-Ordem
1. Lista elementos da subárvore esquerda, 2. Lista elemento corrente, 3. Lista elementos da subárvore direita.

Pós-Ordem
1. Lista elementos da subárvore esquerda, 2. Lista elementos da subárvore direita, 3. Lista elemento corrente.

Elabore diagramas de travessia semelhantes ao apresentado acima para estas duas ordens e obtenha as respetivas sequências de listagem dos elementos. Observe que segundo a ordem **Em-Ordem**, os elementos seriam listados na sequência natural (i.e., por ordem crescente): 3, 5, 6, 9, 14, 23.

A listagem do conteúdo da árvore binária em ordem **Pré-Ordem** pode ser realizada usando os seguintes métodos:

```
// Front-end public
void CArvoreBinaria::escrevePreOrdem() const {
  if(raiz == nullptr){
    cout << "Árvore vazia!" << endl;
    return;
  }
  escrevePreOrdem(raiz);
}
//-------------------------------------------------------------
// Método private
void CArvoreBinaria::escrevePreOrdem(CNoArvore *pArvore) const {
  if (pArvore == nullptr) return;
  cout << pArvore->dados << " ";
  if (pArvore->esq != NULL) escrevePreOrdem(pArvore->esq);
  if (pArvore->dir != NULL) escrevePreOrdem(pArvore->dir);
}
```

Remover elementos de uma árvore binária ordenada é uma operação mais complexa do que a inserção. Tomando como exemplo a árvore da Figura 4, suponhamos que pretendemos remover o nó **1**. Como esse nó é um nó terminal, haveria apenas que destruir o nó (`delete` ...) e guardar no ponteiro `esquerdo` do nó **2** o valor `nullptr`. Igualmente simples é a remoção de um nó com uma só subárvore, pois basta alterar o ponteiro que apontava para o nó removido por forma a que este passe a apontar para a subárvore do nó eliminado (estude o caso da remoção do nó **14**). Quando o nó a remover não satisfaz nenhuma das duas condições anteriores, o processo é um pouco mais complexo.

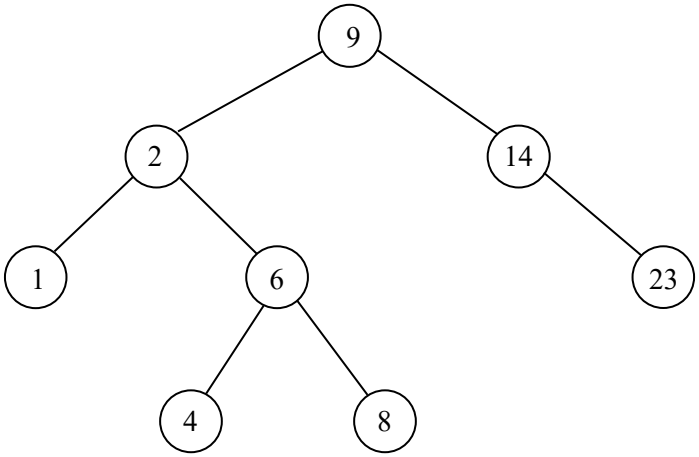


Figura 4

Tomemos por exemplo a remoção do nó **2**. Se apagarmos esse nó sem fazer mais nada, o resultado é o que se ilustra na Figura 5. Falta claramente fazer algo...

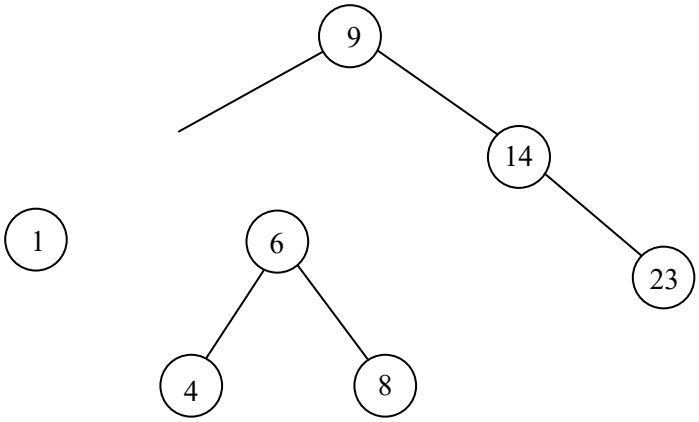


Figura 5

O que está mal nesta situação é que as subárvores esquerda e direita do elemento que foi removido da árvore não foram religadas à árvore. Coloca-se agora a questão: onde colocar estas subárvores, ou os seus elementos? Há várias soluções possíveis. Uma delas consiste em *"Substituir o nó eliminado pelo nó seguinte, segundo a ordem **Em-Ordem**"*. No caso exemplo, o nó que se segue ao nó **2** segundo esta ordem (EmOrdem) é o nó **4**, e o resultado seria o que se mostra na Figura 6.

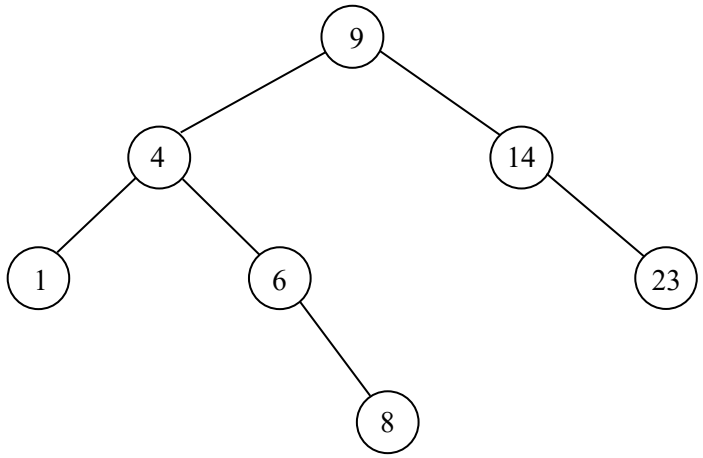


Figura 6

Como pode comprovar, a árvore final é uma árvore binária de pesquisa válida. Existem ainda casos que levantam problemas especiais, tais como a remoção do elemento `raiz`. Na caixa seguinte apresenta-se um método que implementa todos estes casos. Analise o código com cuidado e verifique que permite resolver todos os casos possíveis na remoção de um nó de uma árvore binária de pesquisa...

```
bool CArvoreBinaria::elimina(int item) {
     CNoArvore *pai, *aux, *filho, *sucessor;
     bool encontrado=false;

      // procura o nó a eliminar
     pai = nullptr; aux = raiz;
     while ( (aux != nullptr) && (!encontrado) ) {
         if (aux->dados == item) encontrado = true;
         else if (item < aux->dados) {
             pai = aux;
             aux = aux->esq;
         }
         else {
             pai = aux;
             aux = aux->dir;
        }
     }

     // se for encontrado 'aux' aponta para o nó
     // a eliminar e 'pai' aponta para o seu pai (se existir)
     if (encontrado) {
         // se nó tem no máximo um filho
         if ( (aux->esq == nullptr) || (aux->dir == nullptr) ) {
             if (aux->esq != nullptr) filho = aux->esq;
             else filho = aux->dir;
```

```
            if (pai == nullptr) // caso especial: apaga raiz
                raiz = filho;
            else {           // atualiza ligações
                if (aux->dados < pai->dados) pai->esq = filho;
                else pai->dir = filho;
            }
            delete aux;
        }
        else {  // eliminar nó com dois filhos
            // procura sucessor
            sucessor = aux->dir;
            pai = aux;
            while (sucessor->esq != nullptr) {
                pai = sucessor;
                sucessor = sucessor->esq;
            }
            aux->dados = sucessor->dados; // troca val.
            if (sucessor == aux->dir) aux->dir = sucessor->dir;
            else // substitui pelo seu sucessor `a dir.
                 pai->esq = sucessor->dir;
            delete sucessor;
        }
        return true;
    }
    return false;
}
```

A declaração completa da classe `CArvoreBinaria` é então:

```
class CArvoreBinaria {
   CNoArvore *raiz;
   // ponteiro para a raiz da árvore
   bool procuraRecursiva(CNoArvore*, int) const;
   void escrevePreOrdem(CNoArvore*) const;
   void insere(int, CNoArvore*);
   void destroiSubArvore(CNoArvore*);
public:
   CArvoreBinaria();
   ~CArvoreBinaria();
   bool procura(int) const;
   void escrevePreOrdem() const;
   void insere(int);
   bool elimina(int);
   void destroi();
};
```

10.3. Problemas Sugeridos

Com o conjunto de problemas enunciados a seguir, pretende-se que o aluno consolide os seus conhecimentos relativos a árvores binárias de pesquisa. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa todos os exercícios não realizados durante as aulas práticas. Todos os problemas assumem a declaração das classes `CNoArvore` e `CArvoreBinaria` apresentadas anteriormente e que esta última é estendida para responder ao problema enunciado.

Problema 10.1 – Fácil

Adicione um novo método para listar na consola o conteúdo de uma árvore binária ordenada, segundo a ordem *Por-Ordem*.

Problema 10.2 – Fácil

Adicione um novo método para listar na consola o conteúdo de uma árvore binária ordenada, segundo a ordem *Pós-Ordem*.

Problema 10.3 – Fácil

Adicione um novo método que devolva os valores máximo e mínimo de entre todos os elementos constituintes de uma árvore binária de pesquisa que guarda números inteiros.

Problema 10.4 – Fácil

Adicione um novo método que devolva a soma de todos os elementos de uma árvore binária de pesquisa que guarda números inteiros.

Problema 10.5 – Fácil

Adicione um novo método que devolva a soma de todos as folhas de uma árvore binária de pesquisa que guarda números inteiros.

Problema 10.6 – Fácil

Adicione um novo método que devolva a soma de todos as folhas esquerdas da árvore, ou seja, elementos aos quais se chega através do ponteiro esquerdo dos respetivos nós pai.

Problema 10.7 – Fácil

Adicione um novo construtor da classe `CArvoreBinaria` que inicialize a árvore com os elementos de uma tabela de inteiros passada por parâmetro. Existe um segundo parâmetro, inteiro, que contém o nº de elementos da tabela.

Problema 10.8 – Médio

Adicione um novo método que permita gravar a informação de uma árvore binária de pesquisa num ficheiro, segundo a ordem *Pré-Ordem*. Crie também o método de leitura que permita realizar a operação inversa, ou seja ler o ficheiro e carregar os dados na memória usando uma árvore binária de pesquisa.

Problema 10.9 – Médio

Definindo desequilíbrio de uma árvore binária de pesquisa como a diferença entre a profundidade da folha mais profunda e da folha menos profunda, adicione um novo método que calcule e devolva este valor.

Problema 10.10 – Médio **(saiu no teste de frequência de 23/06/2015)**

Defina o método público `int CArvoreBinaria::profundidade(int item) const` que pesquisa o valor item na árvore. Se o valor for encontrado, o método devolve a profundidade na árvore do nó em que se encontra guardado, ou seja o comprimento do caminho desde a raiz até esse nó. Se não for encontrado, devolve `-1`. O método pretendido não deve chamar nenhum dos métodos já existentes na classe `CArvoreBinaria` e não pode escrever mensagens no ecrã.

Sugestão: Comece por definir um método privado e recursivo que pesquisa o valor numa subárvore e devolve a profundidade do nó, se o valor existir; devolve `-1` se não existir. Para além de `item`, terá de passar como parâmetro a este método auxiliar um ponteiro para o nó na raiz da subárvore.

Problema 10.11 – Difícil **(saiu no exame de recurso de 10/07/2015)**

Considere a classe `CListaInteiros` que é objeto de estudo na ficha 8 das aulas práticas. Defina o método público `bool CArvoreBinaria::caminhoDesdeRaiz(int item, CListaInteiros &cam)` que pesquisa o valor `item` na árvore. Se for encontrado, o método devolve `true` e, através do parâmetro `cam`, passado por referência, devolve a sequência de valores guardados na árvore, ou seja o caminho, desde a `raiz` (valor guardado na cabeça da lista a devolver) até ao nó onde for encontrado o valor pretendido (inclusive). Se não for encontrado, devolve `false` e, através do parâmetro `cam` passado por referência, devolve uma lista vazia (caminho inexistente). O método pretendido não pode chamar nenhum dos métodos já existentes na classe `CArvoreBinaria` e não pode escrever mensagens no ecrã.

Sugestão: Comece por definir um método privado e recursivo que pesquisa o valor numa subárvore e constrói o caminho desde a raiz da subárvore até ao nó onde for encontrado o valor pretendido. Para além de item e da lista `cam`, terá de passar como parâmetro a este método auxiliar um ponteiro para o nó na raiz da subárvore.

Problema 10.12 – Médio **(adaptado do teste de frequência de 16/06/2017)**

Defina um novo método da classe `CArvoreBinaria` chamado `obtemMinExcetoFolhas()`, com visibilidade `public`, constante, que devolve o menor número armazenado na árvore dentre todos os nós que não são folhas da árvore (só contam os nós que tenham descendentes). Se a árvore estiver vazia ou contiver apenas um nó (sem descendentes), deverá ser devolvido o valor `INT_MAX` (o maior dos números inteiros). Para além do método pedido, poderá implementar outros métodos auxiliares que considere úteis para a sua resolução. Os métodos implementados não devem escrever mensagens no ecrã.
Nota: Pretende-se um algoritmo que não tenha de percorrer sempre todos os nós da árvore.

Problema 10.13 – Médio **(adaptado do exame de recurso de 03/07/2017)**

Defina um novo método público da classe `CArvoreBinaria` chamado com a declaração `int valorPai(int n) const` que devolve o número armazenado no nó pai do nó que contém o número `n`. Se `n` estiver na raiz da árvore ou se a árvore estiver vazia deverá ser devolvido o valor `INT_MIN` (o menor dos números inteiros).
Sugestão: Comece por definir um novo método `private` da classe chamado `ponteiroParaPai(..., int n, ...)`, recursivo, que devolve um ponteiro para o nó pai do nó que contém o número `n`. Se `n` estiver na raiz da árvore ou a árvore estiver vazia, este método auxiliar deverá devolver o valor `nullptr`.

Problema 10.14 – Médio **(adaptado do exame de recurso de 03/07/2018)**

Adicione à classe `CArvoreBinaria` os métodos seguintes:

a) Método `CNoArvore* getNo(int, CNoArvore*) const`, com visibilidade `private`, que procura um número inteiro (valor do 1.º parâmetro) numa subárvore – o 2º parâmetro é o ponteiro para a raiz da subárvore –, devolvendo um ponteiro para o nó da subárvore onde for encontrado o número procurado, ou `nullptr` se este não for encontrado na subárvore.

b) Método `int altura(CNoArvore *pArvore) const`, com visibilidade `private`, que devolve a altura do nó na raiz da subárvore passado como parâmetro, ou seja o comprimento máximo do caminho entre esse nó e as suas folhas descendentes. Se a subárvore estiver vazia, o método deve devolver `-1`.

c) Defina o método `int altura(int item) const`, com visibilidade `public`, que devolve a altura do nó da árvore que contém `item`. Se `item` não existir na árvore, o método deve devolver `-1`. O método deve ser implementado com base num algoritmo que consiste na chamada aos dois métodos definidos nas alíneas anteriores.